CARIAL: Cost-Aware Software Reliability Improvement with Active Learning

Boya Sun*, Gang Shu†, Andy Podgurski†, Soumya Ray†
*Google Inc, †EECS Department, Case Western Reserve University
sunboya@google.com, {gang.shu, podgurski, sray}@case.edu

***Abstract*—In the context of field testing (operational testing) of software, we address the problem of balancing the potential reduction in failure risk that developers may achieve by reviewing captured test executions to identify failures (and by successfully debugging their causes) against the cost of reviewing the tests. To achieve a desirable balance, we propose a *cost-sensitive active learning strategy*. Our approach guides developers in selecting a sample of test executions to review and label, and it calls for them to profile execution dynamics and characterize the symptoms and relative severity levels of failures. Profiles, labels, failure symptoms, and severity levels are used by the active learner to construct and refine a mapping between examined and unexamined tests, on one hand, and possible defects, on the other hand. This mapping is used together with estimates of test review costs to guide the selection of additional tests. We evaluate our approach on three subject programs and show that it (1) produces reasonable predictions of risk reduction and (2) significantly improves severity-weighted reliability for each subject program, with relatively low developer effort.**

Keywords-Operational testing, reliability improvement, delivered reliability, active learning, risk reduction, cost-sensitive analysis

I. INTRODUCTION

For a given software system or application, there are often substantial differences among potential test inputs in the *costs* associated with evaluating the input-output behavior they induce. Evaluation of test results typically requires developers either to examine actual output manually or to determine expected output in advance so that it can be compared automatically to actual output. In general, the cost of evaluating a test's results increases with the size and complexity of the test input and of the output it generates. Among test inputs that induce software failures, there are often large differences in the *benefits* obtained, in terms of improved operational reliability, by finding and repairing the software defects they reveal. This is because some defects are triggered more frequently in the field than others and because different defects cause failures of different severities.

Few proposed software testing techniques (see Section VI) are guided explicitly by consideration of the differences in the costs and benefits associated with individual test inputs, perhaps because it is often difficult to predict them prior to testing. However, testers do commonly favor small and relatively simple test cases [31][33], presumably to reduce the effort they must expend in constructing tests, checking the results, and debugging failures. This practice runs the risk of missing defects triggered by more complex inputs from end users. The difficulty of evaluating complex outputs helps explain why developers do not routinely capture inputs in the field [30] when it is feasible, in order to replay and check them or to reuse the inputs for regression testing. It is typical for developers to rely mainly on users to discover and report field failures, though users often perform these tasks poorly [10].

In this paper, we propose a novel, proactive approach to improving software reliability in the context of field testing. It applies to software that has already been deployed, at least for beta testing, and that possibly has also been modified. The approach, which is called *Cost-Aware Reliability Improvement with Active Learning* (*CARIAL*), is intended to balance the costs of reviewing test output against improvements in severity-weighted operational reliability that are predicted to result from fixing the defects the tests may reveal. CARIAL does this by interactively guiding developers, using a *cost-sensitive active learning* technique to investigate an incrementally-chosen sample of software executions, captured in the field, and to provide feedback to the learning algorithm. The algorithm uses the feedback together with execution profiles to construct and refine a mapping between examined and unexamined test executions, on one hand, and possible defects, on the other hand. This mapping is used to empirically estimate the reduction in *operational risk* (defined as expected loss due to failures) that will result if a putative defect is revealed and debugged. Estimates of risk reduction and of test review costs are used to select further tests for developers to review. The former estimates may also be used by developers to help decide whether the software is ready for general release.

CARIAL does not require feedback from end users and it does not require developers to assign specific costs to inputs or to software failures. It does require a means of roughly assessing the *relative* costs of reviewing test outputs, based on properties such as input and output size or complexity. It also requires that a significant and representative sample of complete program inputs be captured in the field. Finally, it requires that developers examine and label selected executions (PASS/FAIL), characterize failure symptoms, and assign levels of *relative severity* to the failures. We argue that in many scenarios, these requirements are reasonable and well-justified by the potential benefits of our approach.

Our work makes the following contributions:

- We propose a novel adjunct to field testing, CARIAL, which balances the cost of test-output examination by developers against predicted improvements in severity-weighted reliability.
- We developed a cost-sensitive active learning algorithm (CostHSAL) to predict, using execution profiles and developer feedback, the risk reduction

that will result from fixing an apparent defect, taking failure rate and severity levels into consideration.

- We report on an empirical evaluation of CARIAL, involving three software projects. The results indicate that CARIAL improves risk reduction, with relatively low developer effort.

The remainder of the paper is organized as follows: Section II gives a brief introduction to the concepts and techniques used in this paper; Section III introduces the CARIAL framework; Section IV describes the CostHSAL algorithm in detail; Section V presents experimental results characterizing the effectiveness of the framework; Section VI summarizes related work; and Section VII concludes the paper and describes future work.

II. BACKGROUND

In this section, we introduce the concepts of field test corpus, failure regions, defect failure rates, and risk from a defect. We also briefly discuss cost-sensitive active learning, the machine learning technique we employ.

A. Field Test Corpus, Failure Regions, Defect Failure Rate

The *operational input distribution* of a software system is the probability distribution of its (complete) inputs in a particular environment [6]. For the purpose of estimating software reliability, it is not necessary to characterize the operational distribution analytically. A random sample of suitable size adequately characterizes the distribution. We assume that a large random sample or corpus of test inputs/executions is captured in the field, in a way that permits replay and review of executions [30]. This *field test corpus* will be sub-sampled by the active learner.

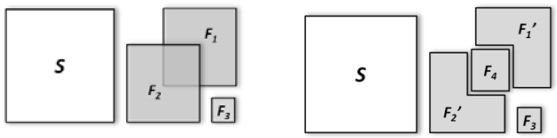
In the field test corpus, there will typically be one *success region* and several *failure regions*. The success region contains all inputs that induce passing executions. Each failure region F_i contains the inputs that induce failures by triggering a particular defect d_i. Let *TC* be the field test corpus. Assuming there are *k* underlying defects, *TC* can be decomposed as follows:

$$TC = S \cup F_1 \cup F_2 \cup \ldots \cup F_k$$

Figure 1a illustrates a test corpus with a success region and several failure regions. Note that the latter may overlap, since an input may trigger several defects and thus belong to several failure regions. In this case, we define new failure regions corresponding to each subset of defects that can be triggered by an input. This makes the failure regions non-overlapping (Figure 1b), and some regions are associated with "superdefects" that are composed of simpler defects. With this modification, we define the *defect failure rate* for a defect (or superdefect) d_i with failure region F_i as

$$\text{Eq. 1: } \boldsymbol{dfr(d_i) = \frac{|F_i|}{|TC|}}$$

The defect failure rate is thus the proportion of operational inputs that trigger a certain defect (or superdefect) d_i. Since failure regions as defined above are non-overlapping, $dfr(d_i)$ is well-defined in that inputs are not double-counted if they trigger multiple defects.



(a) (b)

Figure 1. Illustration of operational distribution of inputs. (a) shows success region and overlapping failure regions; (b) shows success region and non-overlapping failure regions.

B. Risk From a Defect

Intuitively, the risk from a defect depends on two main factors: the defect failure rate and the *severity level* of the defect. Defects with high failure rates cause software to fail frequently, so they should have high risk. On the other hand, defects that are rarely triggered but cause catastrophic failures when they are triggered should also have high risk. The severity level of a defect depends on the nature of the failures it causes. For example, a defect that causes cosmetic errors in output is less severe than a defect that causes a critical system to deadlock. Most bug tracking systems provide predefined severity levels such as *normal*, *major*, and *blocker*.

More formally, we define the *risk of a defect* d_i, $R(d_i)$, as the *expected loss* if d_i remains in the released software. Loss can be measured in different ways. For example, we might assume that each field failure caused by d_i results in a dollar loss $\mathcal{L}(\text{severity}(d_i))$, which depends on d_i's severity level. We then define the risk associated with defect d_i as:

$$\text{Eq. 2: } \boldsymbol{R(d_i) = \mathcal{L}(\text{severity}(d_i)) \cdot dfr(d_i)}$$

The *total risk* from all defects in a program will thus be

$$\text{Eq. 3: } \boldsymbol{R_{tot} = \sum_i R(d_i)}$$

The goal of our work is to reduce the total risk in the program as much as possible while expending as little developer effort as possible in reviewing test case outputs.

C. Cost-Sensitive Active learning

For some problems, it is very expensive and difficult to obtain class labels [28]. An *active learning* algorithm obtains labels from an "oracle", such as a human user, interactively and incrementally. At each step, the algorithm selects examples that are predicted to most improve the current model, if their labels were known. For example, a common strategy is to pick examples the current model is most uncertain about. Since (mostly) "useful" examples are labeled, fewer labels should be needed to produce an accurate model than for standard supervised learning.

We assume that for most software products, the cost of an operational failure is more variable and potentially much higher than either the cost of reviewing a test or the cost of debugging a defect. Moreoever, debugging a given defect may require examining multiple failing tests. Hence, when a

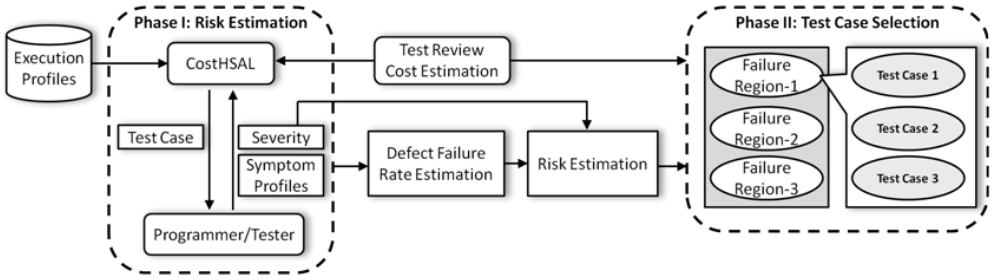


Figure 2. Framework of CARIAL. Phase I constructs a mapping from failures to hypothesized defects and uses it to estimate risk. Phase II selects test cases to debug the underlying defects. In each phase, the framework attempts to minimize the cost of reviewing test cases.

failure is discovered during review of test outputs, it is often reasonable to defer debugging until the relative risk posed by different defects is clarified by reviewing additional tests.

Since examining test results is expensive, we use active learning to efficiently guide exploration of the field test corpus and to construct a *risk mapping* that relates observed and unobserved failures to underlying defects. Our approach is based on an existing active learning algorithm, called *hierarchical sampling for active learning* (HSAL) [7]. This algorithm, like many other active learning algorithms, tries to minimize the *number* of labels needed while maximizing the accuracy of the model, assuming that every label is equally expensive to obtain. This assumption does not hold in our case, where it can be significantly more expensive to examine complex outputs than simple ones. Hence we modified the HSAL algorithm to make it cost-sensitive, that is, to minimize the sum of (non-uniform) costs of obtaining labels.

III. THE CARIAL FRAMEWORK

The CARIAL framework is illustrated in Figure 2 and consists of two phases. The goal of the first phase is to estimate, as accurately as possible, the total risk due to defects. To do this, CARIAL selects test cases for developers to examine. This step uses active learning to try to balance the cost of reviewing test case outputs against the potential improvement in risk estimation. Developer feedback is used to update defect failure rate estimates and severity information. In the second phase, these are used to select test cases that can be used to debug defects. These test cases are chosen by the algorithm to maximize total risk reduction while minimizing review costs. In the following, we describe each phase of the framework in detail. We assume that *crashing tests* are identified automatically and are processed by developers *prior* to active learning. Hence the subsequent discussion will assume that observed failures do not cause crashes.

A. Phase I: Estimating Total Risk with Active Learning

To estimate the risk posed by a defect d_i, it is necessary, according to Equation 2, to estimate its severity and its failure rate. Estimating either quantity might seem to require first identifying d_i in program code, in order to determine which failures it caused on the test corpus. This would put the cart before the horse, so to speak, since the traditional purpose of testing is to induce failures that reveal defects, so that the latter can be identified and corrected. To address this issue, one possibility is to randomly and uniformly sample tests from the input corpus for developers to review, and to construct approximations of the success and failure regions based on their feedback. However, this is unlikely to make efficient use of their time and effort, because simple random sampling does not make use of what is already known about different regions of the corpus, so a lot of time could be spent reviewing tests that do not add any significant information. For example, if severe failures are rare then simple random sampling of tests may cause developers to expend most of their effort on reviewing low risk tests. Active learning, on the other hand, can make use of information obtained from already sampled units, in order to maximize the information gained through further sampling while limiting the developer effort that is required. In CARIAL, we exploit this idea.

1) Learning algorithm in a nutshell

The cost-sensitive active learning algorithm we use, CostHSAL, is described in detail in Section IV, but we outline the key ideas here. CostHSAL uses execution profiles collected from tests to construct a cluster-based model of failure regions and of the success region. The profiles may be any of the types used in software testing to characterize execution dynamics, including profiles of function coverage, statement coverage, branch coverage, data flow coverage, etc. Initially, unlabeled regions of the input corpus are identified by hierarchical cluster analysis applied to execution profiles alone. At each subsequent iteration, CostHSAL selects an unexamined test, based on the current model, for a developer to review. The choice of test is based on the size and homogeneity of the current estimated failure and success regions and on the assessed costs of reviewing the tests they contain. After reviewing the output of a test, the developer labels the test as a success or a failure, and in the latter case he/she characterizes the failure's severity and its externally observable symptoms (see below). This is used to update estimates of failure rates and failure severity for defects. We assume there is a *cost budget*, which is the total cost a developer is willing to spend reviewing test outputs. The active learning iteration

stops either when this budget is exhausted or when the algorithm decides the clusters are sufficiently "pure".

In determining which captured field-tests developers should review, CostHSAL considers the costs of examining the output produced by individual tests. These costs may be considerable, especially for software that produces large or complex outputs, because developers typically must examine outputs manually. For example, if a program outputs graphs, developers are likely to spend more time evaluating dense graphs than they spend evaluating sparse ones. We assume that the average cost of examining the output of a test grows proportionately with a measure of the size or complexity of the test. We currently employ linear cost metrics, e.g.,

$$cost(t_i) = \alpha \cdot |output(t_i)|$$

$$cost(t_i) = \alpha \cdot complexity(output(t_i))$$

where α is a constant, $|output(t_i)|$ is the size of the output produced by test t_i, and $complexity(output(t_i))$ is a measure of the complexity of t_i's output. We do not currently consider debugging costs, because of the difficulty of predicting them.

When CostHSAL terminates, test cases are grouped into an estimated success region $\hat{S}$ and several estimated failure regions (clusters) $\hat{F}_i$. Failures with distinct symptoms are placed in different failure regions. Further, if one test case triggers a set of defects rather than a single one (see Section II.A), we assume that the symptoms will be determined by the whole set of defects, so that test cases triggering different sets of defects have different symptoms. Since each failure has one set of symptoms, the regions $\hat{F}_i$ will be *non-overlapping failure regions* (NOFRs). We will discuss the consequences of the assumptions not holding in Section V.E. Note that our assumptions about test cases triggering multiple defects mean that such test cases will be allocated to their own groups, and we will see them as being caused by a new "superdefect" (composed of simpler defects). This will have its own severity level as determined by the developers. This is possibly suboptimal in that it would be better to associate those test cases somehow with the individual defects. However, it is not obvious how to do this, since at this stage we do not even know how many defects are present in the program, or where they are.

2) Developer feedback

CARIAL requires developers to provide two pieces of information about a failed test (in addition to labeling it as a failure): a severity level and a symptom profile. The *severity level* may be specified on an ordinal scale or by a number. The simplest scheme is to use a fixed set of levels like those used in bug tracking systems. Developers must also specify a loss function $\mathcal{L}$ to map severity levels to numbers. It is not necessary to assign an exact dollar cost to each failure. It suffices to roughly characterize the *relative costs* of different types of failures, e.g., to within an order of magnitude. CARIAL currently considers only the severity levels of failures actually observed by developers. This may lead to inaccurate risk predictions if rare failures not triggered by the field test corpus lead to catastrophic losses. These could be addressed using long-tailed parametric models for the probability distribution of severity levels.

A *symptom profile* characterizes the symptoms of a failure. Failed tests that have similar symptom profiles will be assigned to the same failure region. Symptom profiles can be provided by developers as attribute vectors or as machine-checkable signatures. Symptom vectors can be obtained from users with a well-designed *user feedback mechanism* that allows them to quickly answer (e.g., with mouse clicks) a set of pre-defined, application-specific questions about a failure's symptoms. For example, the *JavaPDG* program dependence graph generator [20] developed by our group provides such a mechanism, which questions the user about erroneous graph elements. Since the questions are predefined, it is easy to extract the feedback into a feature vector.

Alternatively, symptom profiles can be provided using a *symptom signature*, which is a machine-checkable property or set of properties characterizing a set of related failures. For example, we use *ROME* [26], an RSS reader, as a subject program. The symptom of one particular defect can be described as "<href> is a relative URL instead of an absolute one", which is an easily checkable property. Burger *et al* [4] used similar ideas in the context of automatic debugging. They inserted *predicates* into the code to detect symptoms of a failure, such as "attribute name of object with id 13 has value "UTC"".

B. Phase II: Selecting Test Cases for Debugging Defects

At the end of the first phase, we have a set of failure regions corresponding (ideally 1:1) to defects in the program. In the second phase, we use these to select test cases from programmers to use to debug these defects. Here, we assume that developers may need to examine multiple tests that trigger a given defect in order to successfully debug it. The goal is to select test cases that (i) minimize cost to review and (ii) maximize the reduction in risk, if the defects associated with them can be debugged. In each iteration the algorithm determines the failure region with the currently maximal expected risk. From this region, we repeatedly select test cases in increasing order of review cost. Once developers have identified enough failing tests from the maximal-risk region to permit the underlying defect(s) to be successfully debugged, we move on to the next riskiest region and repeat the process.

As an optimization in this phase, developers may reuse previously labeled test cases from Phase I. These test cases will have review-cost zero in this phase, since their output has already been examined earlier.

IV. The Cost-Sensitive Active Learner: CostHSAL

In Phase I of CARIAL, we use a cost-sensitive active learning algorithm, CostHSAL. This algorithm is a modified version of an algorithm presented in prior work, Hierarchical Sampling for Active Learning (HSAL) [7].

Algorithm 1: CostHSAL

Input: hierarchical clustering of test cases; batch size *B*; budget *G*
Output: Pruning (cut) of cluster tree annotated with labels

$P \leftarrow$ {root} [current pruning of tree]
repeat until budged exhausted or *P* is pure
  for ***i* = 1 to *B***
    $v \leftarrow$ `SelectCluster`(*P*)
    tc $\leftarrow$ `SampleTestCase` (*v*)
    SymptomProfile, *Severity* $\leftarrow$ `QueryDeveloper`(*tc*) [active learning query; *SymptomProfile* and *Severity* are NULL if *tc* triggers no failure]
    Update label statistics of *v* according to *SymptomProfile*
  end for
  for all impure clusters v_i in *P*
    $P \leftarrow P \setminus \{v_i\} \cup \{\text{children}(v_i)\}$
  end for
end repeat
For each cluster $v_i \in P$, (1) assign all members the majority label, (2) assign its severity as the average severity among all members
return label-annotated pruning

Algorithm 2: SelectCluster

Input: Current pruning *P* of cluster tree
Output: Cluster to sample test case from

for each cluster v_i in *P*
  $\omega_i = |v_i|/N$ [*N* is the total number of test cases]
  $p_i =$ proportion of majority label out of all labeled nodes in v_i
end for
Sort clusters by $\omega_j\ (1 - p_j)$ into non-increasing order
$c^{min} \leftarrow \infty$; $v^{min} \leftarrow$ NULL
repeat
  Remove *v* from front of sorted cluster list
  $c \leftarrow$ `MeanSampleCost`(*v*)
  if $c < c^{min}$ **then** $c^{min} \leftarrow c$; $v^{min} \leftarrow v$
  else return v^{min}
end repeat

HSAL has two very desirable properties for our problem. First, many active learning algorithms apply to classification problems for which the number of classes is known. In our case, "classes" correspond to defects, so we have no idea how many classes there are. Fortunately, HSAL can start with only one class, and when the user provides a new class label, it can update the number of classes. Second, HSAL can allocate multiple regions for the same class. This can be useful because different test cases, though they all share the PASS/FAIL label, can have very different feature values. For example, the ROME project uses different classes and functions to parse different types of feeds, such as RSS 1.0, RSS 2.0, and Atom 1.0. Test cases that involve different types of feeds can have very dissimilar code-coverage profiles, and HSAL can take this into account. However, the basic HSAL algorithm does not incorporate non-uniform costs for label acquisition. We next describe HSAL, followed by our modifications to make it cost-sensitive.

A. The Basic HSAL Algorithm

HSAL is an active learning algorithm that exploits cluster structure in data. It takes as input a hierarchical clustering of the unlabeled dataset, which could be generated by any chosen hierarchical clustering algorithm. At each step, the algorithm maintains a *pruning* (cut) of the cluster tree which is a partition of the data set. The goal of the algorithm is to quickly (with little labeling effort) reach a pruning where the constituent clusters are fairly pure in their class labels.

The pseudocode of the algorithm is shown in Algorithm 1. The initial pruning that HSAL starts with consists of just the root, v_1. In each iteration, the algorithm selects some cluster from the current pruning (`SelectCluster` in the Algorithm 1), then probabilistically traverses the subtree below the chosen cluster and samples a currently unlabeled leaf (`SampleTestCase`). It then queries the oracle to obtain its PASS/FAIL label, and if the test fails, obtain its symptom profile and severity level (`QueryDeveloper` in the Algorithm 1); the symptom profile serves as its class label. With this label, HSAL updates statistics that summarize the frequency of different classes in each cluster. It uses these summary statistics to identify clusters that are *impure*, i.e., composed of different classes. It then removes these clusters from the current pruning and replaces them with their children, which intuitively may be expected to be less impure. This process results in an updated pruning. The algorithm continues until the clusters in the current pruning are fairly pure or until the user-defined cost budget is exhausted.

To select a cluster from the current pruning, HSAL samples from a probability distribution where a cluster v_i's likelihood of being chosen is proportional to $\omega_i\ (1 - p_i)$, where $\omega_i = |v_i|/N$ is the fraction of all points allocated to v_i and p_i is the proportion of all labeled points in v_i having the majority label in v_i. The closer that p_i is to 1, the more pure v_i is. This criterion makes it more likely for HSAL to pick large, impure clusters. Then, to pick an individual point to query, the algorithm traverses the subtree below the selected cluster. Here, at each step, it picks a child with probability proportional to the size of unlabeled data in the child. This process is repeated until a leaf node is reached and returned. This focuses HSAL's labeling effort on parts of the space that are still largely unexamined.

We illustrate the algorithm using the example cluster tree in Figure 3 which might be the result of hierarchically clustering a set of items. At the leaves of this tree we show the fraction of examples with a certain label. Assume this is a two-class problem. The leftmost leaf has 20% of all the data, labeled with one class (colored white). The next leaf contains 30% of the data, all labeled with the other class (colored grey), and so forth. Consider the cluster tree in Figure 3. Initially, we have $P = \{v_1\}$. Half the data in this node are from one class and half from the other (seen by traversing the subtrees below it), so after sampling some points and determining their labels, v_1 should be found to be impure, so HSAL moves to pruning $\{v_2, v_3\}$. Depending on how the labeled points are distributed in v_2 and v_3, one of them will be judged to be more impure and (probabilistically) selected to

Algorithm 3: MeanSampleCost

Input: Cluster *v* from cluster tree
Output: Estimate of average cost to query node in *v*

if *v* is a labeled leaf **then** return ∞ [ensure these cannot be selected]
if *v* is an unlabeled leaf **then** return *cost*(*v*) [cost functions defined in III.1]
$s \leftarrow 0$
for a constant number of iterations *k*
  testcase ← `SampleTestCase` (*v*)
  $s \leftarrow s$ + *cost*(*testcase*)
end for
return *s*/*k*

Algorithm 4: SampleTestCase

Input: Cluster *v* from cluster tree
Output: Test case from subtree below *v*

if *v* is a leaf node **then** return *v* [leaves are test cases]
else
  Select a child *c* of *v* with probability
    $\Pr(c) \propto \frac{1}{\text{MeanSampleCost}(c)}$ [prefer low cost children]
  return SampleTestCase(*c*)

sample from. More labels will reveal that v_2 is quite impure and needs to be split. This continues until the label budget runs out or a pure pruning is found.

B. CostHSAL

For our problem, we need to make HSAL cost-sensitive, so that it takes into account the fact that different tests have different review costs. This can be achieved by modifying the two functions `SelectCluster` and `SampleTestCase` to be sensitive to test review costs. The basic algorithm remains the same as HSAL. The modified functions are shown in Algorithms 2 and 4**.** These use an auxiliary function, `MeanSampleCost` (Algorithm 3), which estimates the average cost of unlabeled test cases in a cluster. To modify `SelectCluster`, we would like to select not just proportional to $\omega_i\,(1-p_i)$ but also *inversely* proportional to the mean sampling cost of the cluster, so that clusters that are large and impure but also relatively inexpensive to label are selected. An intuitive approach is to select a cluster *v* with probability proportional to $\frac{\omega_i\,(1-p_i)}{\text{MeanSampleCost}(v)}$. However, since $\omega_i\,(1-p_i)$ is a probability and `MeanSampleCost`(*v*) is not, it is hard to put these two metrics on the same scale. Hence we employ a heuristic scheme based on sorting clusters by $\omega_i\,(1-p_i)$ into non-increasing order and then walking the sorted list until the cost starts to increase. We then return the cluster with the lowest cost found. To modify `SampleTestCase`, we use an iterative procedure as in HSAL, except that in each iteration, instead of selecting children just on the basis of the fraction of unlabeled examples, we select a child *c* with probability inversely proportional to `MeanSampleCost(c)`, so that a child with low mean sampling cost is more likely to be selected.

V. Empirical Evaluation

We perform an empirical evaluation to assess the effectiveness of the CARIAL framework for balancing risk reduction and test review costs. This study addresses two main research questions:

- RQ-1: How accurate is the risk estimate that is output by Phase I of CARIAL?

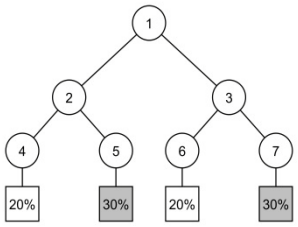


Figure 3. Example cluster tree

- RQ-2: How effective is the test case selection scheme used by Phase II of CARIAL?

To answer these questions we evaluated CARIAL and two other baselines on several subject programs.

A. Subject Programs

We used three open source projects in our evaluation: JavaPDG [20], ROME [26] and Xerces2 [35], each of which produces complex output. JavaPDG [20] is a software tool developed by our research group to input a Java program and output a System Dependence Graph [17], in which the vertices represent program elements, such as declarations, expressions or predicates, and the edges represent data and control dependences. ROME [26] is an open source Java library for parsing, generating, and publishing RSS and Atom feeds. We constructed a driver program that uses the ROME APIs to parse RSS and Atom feeds and output them in XML. Xerces2 [35] is an open source Java XML parser. We wrote a driver for Xerces2 that takes an XML file as input, parses it, and then assembles the parsed elements into a new XML file. The characteristics of the subject programs are summarized in Table I.

Test Cases. We collected operational inputs for each of the three projects. For JavaPDG, we used functions from the *Spring Framework* [29] as inputs; altogether 1295 test cases were collected. For the ROME and Xerces2 projects, we reused test cases from Augustine et al's work [1]: for ROME, 8,000 Atom and RSS files were downloaded from Google Search results, using a custom web crawler; for Xerces2, 9,630 files were collected from the system directories of an Ubuntu Linux 7.04 machine and from Google Search results.

Defects. We selected a sample of defects from the bug database of each project. Nine defects were randomly selected for JavaPDG, six for ROME, and eight for Xerces2. For each of the defects, custom code instrumentation was inserted to detect its failure conditions and report when they were triggered. The JavaPDG defects caused 3.9% test cases

TABLE I. Summary of Subject Programs Used in The Empirical Evaluation.

"**# Test Cases**" refers to number of test cases; "**%failures**" refers to the fraction of failing test cases, "**#Defects**" refers to the number of true defects, "**#NOFRs**" refers to Non-Overlapping Failure Regions and "**Cost Function"** refers to the assumed cost of reviewing test cases.

Program	#Test Cases	%failures	#Defects	#NOFRs	Cost Function
JavaPDG	1295	3.9%	9	12	$\alpha \cdot$ #Edges
ROME	5425	5.2%	6	13	$\alpha \cdot$ Size of Output XML
Xerces2	4773	5.4%	8	14	$\alpha \cdot$ Size of Output XML

to fail, while the ROME and Xerces2 defects caused more than 30% of their test cases to fail. In typical field testing scenarios, it is unlikely that 30% or more of executions will fail. Hence, we randomly discarded failing tests to reduce the failure rates to 5.2% for ROME and to 5.4% for Xerces2. We used a single-linkage based agglomerative clustering algorithm [16] to compute the hierarchical clustering of test case profiles and we manually inspected the resulting cluster tree. For each of the projects, there were test cases that triggered multiple defects, so there were overlaps in the failure regions. We found 12, 13, and 14 non-overlapping failure regions (as described in Section II.A and III.A) in the JavaPDG, ROME and Xerces2 test sets, respectively. We assigned high severity levels to defects with low failure rates and assigned low severity levels to defects with high failure rates. This was done to simulate a difficult scenario in software testing in which the most important bugs manifest very rarely, making it hard to identify and reduce operational risk. For failure regions with multiple defects, we set the severity level to be the highest among the underlying defects.

Execution Profiles. We used *function coverage* profiles in our study, to characterize internal execution dynamics. We used the Java Interactive Profiler [19] to record the number of times each function was invoked during an execution, and we added binary indicators to the profiles to indicate which functions were executed at least once per run.

Symptom Profiles. We used JavaPDG's built-in user feedback mechanism to collect symptom profiles. For ROME and Xerces2, we used *symptom signatures* as symptom profiles. These were provided as text strings describing a property of the output. For all three projects, symptoms were distinct for non-overlapping failure regions.

B. Methodology

1) Cost Functions

As stated in Section III.A, we assume that the cost of reviewing test output is linear in the size or complexity of the output. For JavaPDG, we assumed that this cost is linear in the number of edges of the system dependence graph produced. For ROME and Xerces2, we assumed that this cost is linear in the size, in bytes, of the XML output files.

2) Risk Estimation

In our study, we used the built-in importance levels of the Bugzilla bug tracking system [5] as our severity levels. There are six pre-defined importance levels, namely *trivial*, *minor*, *normal*, *major*, *critical* and *blocker*. We represented these levels by the numbers 1 to 6, respectively.

We defined the loss function $\mathcal{L}$ for a defect of severity s as

$$\text{Eq. 4: } \mathcal{L}(s) = M \cdot 2^s$$

where M is a dollar multiplier. We used M = \$1000 in this work.

3) Study Parameters

One parameter of our study design is the cost budget for risk estimation, denoted by B. In an actual application of CARIAL, the parameter B is specified by developers in order to control the effort they expend on risk estimation. Obviously, B affects the accuracy of risk estimation. Another parameter is the mean of the number X of failures, caused by a particular defect d_i, that developers need to review in order to successfully debug d_i. We assume that the number X is not fixed but varies around some typical number of test cases. Thus we model X in our study as a Poisson random variable with mean λ.

4) Baseline Approaches

We compared CARIAL against two baseline approaches to test selection: simple random sampling from the test corpus and smallest-first sampling. *Smallest-first sampling* always selects an unlabeled test case with lowest review cost. It arguably mimics what developers often do in practice. For a fair comparison, we adapted these baselines to also operate in two phases, a risk estimation phase and a test case selection phase.

In the risk estimation phase, random sampling and smallest-first sampling were each used to choose a sample T of tests for developers to review. Then, test cases in T with the same symptom profiles were grouped into non-overlapping failure regions just as with CARIAL. The risk was estimated using Equation 2 with the same loss function shown above.

In the test case selection step, we used the optimization described for CARIAL in Section III.B, where test cases labeled in Phase I are first used to debug defects. When they were exhausted, random sampling or smallest first sampling was used to continue selecting test cases to debug defects.

C. RQ-1: Accuracy of Risk Estimation

We define the Error of Risk Estimation as the absolute difference between estimated total risk and the true total risk:

$$ERE = \left|\widehat{R_{tot}} - R_{tot}\right| = \left|\sum_i \hat{R}(\hat{d}_i) - \sum_j R(d_j)\right|$$

Since Phase I of CARIAL outputs non-overlapping failure regions, each assumed to be associated with a defect, the first term is computed by summing over all failure regions. To compute the second term, we used our knowledge of the true defects and which test case triggers which defect. We used

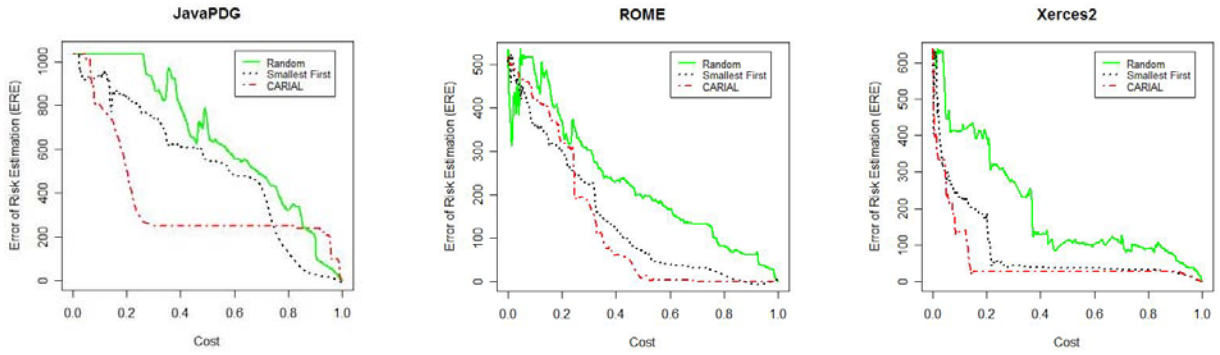


Figure 4. RQ-1: ERE against Cost Curves. Red-CARIAL; Green-Random Sampling; Black- Smallest First Sampling.

this to construct the true non-overlapping failure regions, and summed over those to obtain the true total risk. To evaluate the accuracy of risk estimation, in Figure 4 we plot *ERE* against test review cost, as a proportion of the maximum possible cost, for CARIAL, random sampling, and smallest first sampling, for all three projects. The *x*-axis of the curve represents percentage of total cost spent so far, so the constant α used in the cost function does not matter. The results are shown in Figure 4.

From these results, we first observe that simple random sampling is a poor risk estimator as a function of labeling effort. Further note that, while we show one result for random sampling, there is likely to be wide variation between different runs for this baseline. Second, in some cases, smallest-first sampling is a good risk estimator. This may happen, for example, when most of the test cases are hand-written and produce simple outputs. However, this is not always the case, as shown by the result for JavaPDG. Further, it is important to note that since smallest-first and random sampling do not inherently estimate risk or use it to guide their choices, it may be difficult to tell during execution how accurately the risk is being estimated and how much value we are receiving for our labeling effort. Finally, we observe that CARIAL is generally a good risk estimator—it is usually able to quickly construct an accurate model of the failure regions and estimate the total risk, using less than 20% of the maximum cost.

D. RQ-2: Effectiveness of Test Case Selection

To evaluate effectiveness of test case selection, in Figure 5 we plot the true total risk against cost as more test cases are reviewed. In these graphs, we restrict the *x*-axis (the review cost in this phase) to 20% of the total cost. As described in Subsection B.3 above, the performance of test case selection is affected by two parameters: the review cost budget B of Phase I, as a fraction of the total possible cost, and the mean λ of the Poisson distribution that determines how many test cases a developer might need to see in order to debug a defect. We do not include the cost of the actual debugging in these results. First, it is difficult to predict, and second, it seems plausible that it will be similar for each of our methods, so that the *relative* differences are primarily governed by the costs of reviewing test case outputs. We plot the curves using different combinations of values of B and λ. For B, we choose three values, 0.05, 0.10, 0.15. This reflects our belief that developers may not be willing to spend much effort on risk estimation by itself, since it does not lead to immediate risk reduction. For λ, we chose two values, 3 and 10, to show how the three approaches behave when the defects are "easy" to debug and "hard" to debug. Due to space constraints, we show only the six curves for the JavaPDG project in Figure 5. The behavior of the other programs is similar.

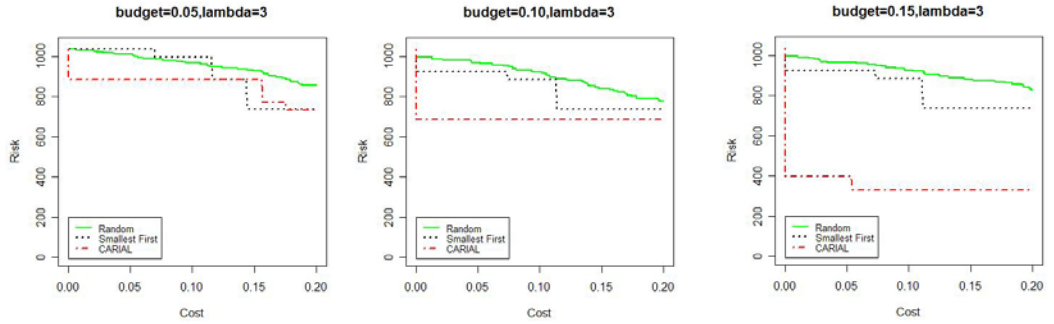
From these results we first observe that CARIAL is generally effective at selecting test cases that quickly reduce the total risk. Note the large vertical drop at the very beginning of the curves is a result of the optimization where the test cases labeled in Phase I are being used first to find defects. These test cases have zero cost in Phase II. These drops are more significant for CARIAL than for the baseline techniques, indicating that the test cases used for risk estimation are also very good at characterizing the underlying defects.

Effect of λ. Comparing Figures 5a and 5b, we observe the CARIAL framework is less likely than the baseline techniques to be affected by high values of λ. This is because CARIAL produces a partitioning of test cases with relatively good accuracy; therefore multiple test cases triggering a defect can often be retrieved from the corresponding failure region. The baseline approaches lack such a partitioning, and so are more affected. For example, when B is 0.10, the black curve representing smallest-first sampling and the red curve representing random sampling are "lifted" quite a bit when λ increases to 10; on the other hand, the red curve representing CARIAL stays roughly stable for both λ values.

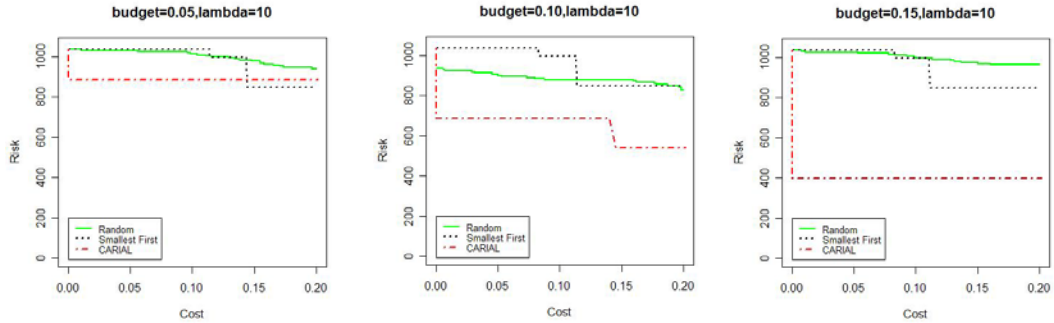
Effect of B. For any λ, comparing the graphs for different B shows that if more effort is spent during risk estimation, it pays off significantly in Phase II for CARIAL. This is because with a larger budget, CostHSAL can better predict the success and failure regions, and thus find a more precise risk estimate. The baselines are relatively unaffected by changes in B, as might be expected since they are not guided by risk estimates in any way when selecting test cases.

E. Threats to Validity

The cost function and the loss function used in the evaluation might be unrealistic or omit some important factors. In future work we will investigate alternative models for the cost of testing and for loss due to software



(a) $\lambda = 3$, $B = \{0.05, 0.10, 0.15\}$



(b) $\lambda = 10$, $B = \{0.05, 0.10, 0.15\}$

Figure 5. RQ-2: The Cost-Against-Risk-Reduction Curves for JavaPDG with different B and λ. Red-CARIAL; Green- Random Sampling; Black- Smallest First Sampling

failures. Moreover, although the defects we collected in all three projects exhibit distinct failure symptoms, there are cases in which a defect triggers failures with different symptoms or two defects trigger failures with the same symptoms. Such defects will be considered in future studies.

VI. Related Work

A. *Cost-Sensitive analysis in software testing and reliability*

The cost of testing and of software failures have been considered in previous research on reliability estimation and test resource allocation. Pham et al [23] proposed a cost model together with a reliability growth model. The cost model, which is linear, incorporates the cost of testing, fault removal and fault risks. Gokhale et al [13] consider the problem of maximizing reliability with given amount of testing effort. Huang et al [18] assume that a reliability objective is given, and aim to achieve an optimal allocation of testing effort to software modules. The costs addressed in [13] and [18] are associated with software modules and software development process. By contrast, our approach considers test case review costs and the failure costs (risk) associated with estimated failure regions. Brown et al [3] seek to balance the cost of testing and the cost of defects by determining an optimal number of software test cases. In contrast to our work, the cost of testing is assumed to be linear in the number of test cases, and severity is not considered when estimating the cost of a defect.

Tsoukalas et al [32] presented estimators and confidence bound formulas for expected failure cost per execution. Gutjahr [14] generalized input-domain based reliability measure proposed previously by introducing expected failure cost, also called risk, as a measure of software reliability. Weyuker [34] refined an approach to load testing proposed by a previous work, which is based on a characterization of the operational distribution of a system's workload, to consider the cost of failures. In contrast to CARIAL, each of these approaches lacks of a concrete method for identifying and characterizing risk regions in conjunction with testing.

Cost has also been addressed in regression testing research. Leung et al [21] proposed a cost model to compare the cost and benefits of selective regression testing strategies against the traditional retest-all strategy. Malishevsky et al [22] present models to compare cost-benefit tradeoffs of test case selection, reduction, and prioritization. Rosenblum et al [27] proposed a model to predict cost-effectiveness for selective regression testing with use of coverage information. In these papers, unlike in our work, cost is assumed to be uniform among test cases, and severity levels are not considered. Elbaum et al's work [9] takes differences in test cost and fault severities into consideration in test case prioritization, and it orders test cases according to *unit-of-fault-severity-detected-per-test-cost-unit*. Their work is similar to ours in that it considers the tradeoff between test costs and failure costs; however, theirs does not deal with the problem of estimating such costs.

B. *Test case partitioning*

A partitioning of test cases can be used to estimate reliability, identify failures, or reduce the size of the test set. Podgurski et al [25] proposed the use of stratified sampling, based on cluster analysis, for estimating software reliability.

Dickinson et al [8] applied cluster analysis to discover failures induced by a large set of test cases. Bowring et al [2] proposed to use active learning to classify program behavior as passing or failing. Podgurski et al [24] proposed automated support for grouping similar failure reports from users, in order to reduce the number of executions developers need to review. Later work by Francis et al [11] proposed a tree-based approach to refine the failure groups. Unlike our research, the aforementioned research does not consider differences in the review costs of individual tests or defect severity. Moreover, [24] and [11] group only failures, while in our work, we classify a mixture of passing and failing test cases, and the status of the test cases is unknown before manual examination of the test outputs. This is a much harder problem, especially because the proportion of failures may be very small.

VII. CONCLUSIONS

We have presented an approach to improving software reliability through explicitly estimating and reducing the risk in a program due to different defects. We do this through active learning. This also allows us to minimize test case review costs at the same time. An empirical evaluation indicated that our approach estimates and reduces risk effectively and at a relatively low cost compared to two baseline techniques.

Many interesting directions remain to be investigated. We plan to investigate alternative cost and loss functions and symptom profiles. Other kinds of costs, such as fault localization costs or test case execution times, could be considered for minimization. Finally, the risk mapping returned by Phase I of CARIAL warrants further study. Currently we are using this just to select test cases. However, it may be useful in other ways, such as for identifying input regions to explore through test case creation.

REFERENCES

[1] V. Augustine, "Exploiting User Feedback to Facilitate Observation-based Testing," Ph.D. Dissertation, EECS, CWRU, 2009.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," In Proc. of ISSTA, Jul, 2004, pages 195 – 205.

[3] D. B. Brown, S. Maghsoodloo, and W. H. Deason, "A cost model for determining the optimal number of software test cases," IEEE Transactions on Software Engineering vol. 15(2), 1989, pp. 218–221.

[4] M. Burger, A. Zeller. "Minimizing Reproduction of Software Failures ", In Proc. of ISSTA, Toronto, Canada, Jul., 2011.

[5] Bugzilla. http://www.bugzilla.org/

[6] B. Cukic. and F. B. Bastani, "On reducing the sensitivity of software reliability to variations in the operational profile," In Proc. of ISSTA, 1996, pp. 45-54.

[7] S. Dasgupta and D. Hsu. "Hierarchical sampling for active learning," In Proc. of the 25th ICML, 2008, pp. 208–215.

[8] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," In Proc. of 23rd ICSE, Toronto, May 2001, pp. 339-348.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," In In Proc. of ICSE, pages 329–338, May 2001.

[10] M. R. Fine, "Beta Testing for Better Software," Wiley, 2002.

[11] P. Francis, D. Leon, M. Minch, A. Podgurski, "Tree-Based Methods for Classifying Software Failures," In Proc. of the 15th ISSRE, Nov., 2004, pp.451-462,

[12] P. G. Frankl, R. G. Hamlet and B. Littlewood, "Evaluating Testing Methods by Delivered Reliability", Transaction on Software Engineering, Vol. 24, August, 1998. pages 586-601.

[13] S. Gokhale. "Cost–constrained reliability maximization of software systems". In Proc. of Annual Reliability and Maintainability Symposium, Los Angeles, CA, January 2004, pages 195-200.

[14] W. J. Gutjahr. "Optimal test distributions for software failure cost estimation," Software Engineering, IEEE TSE on 1995, pp. 219-28.

[15] R. A. Haertel, K. D. Seppi, E. K. Ringger, and J. L. Carroll. "Return on investment for active learning," In Proc. of the NIPS Workshop on Cost-Sensitive Learning, 2008.

[16] hclust (hierarchical clustering package used in R): http://stat.ethz.ch/R-manual/R-patched/library/stats/html/hclust.html

[17] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graph," ACM Trans. Program. Lang. Syst. 12, 1, Jan, 1990, pages 26-60.

[18] C.Y. Huang, J.H. Lo, S.Y. Kuo, M.R. Lyu, "Optimal allocation of testing-resource considering cost, reliability, and testing-effort", In Proc. of the10th IEEE Pacific Rim International Sympsium on Dependable Computing, 2004, vol. 3-5, 2004, pages. 103 – 112.

[19] Java Interactive Profiler. http://jiprof.sourceforge.net/.

[20] JavaPDG Project. http://selserver.case.edu:8080/javapdg/index.htm

[21] H. K. N. Leung, L. White, "A Cost Model to Compare Regression Test Strategies, " In Proc. of ICSM., pages 290–300, Nov. 1990.

[22] A. G. Malishevsky, G. Rothermel, and S. Elbaum, "Modeling the Cost-Benefits Tradeoffs for Regression Testing Techniques," In Proc. of ICSM, Montreal, Quebec, Canada, Oct. 2002, pages 230-240.

[23] H. Pham, X. Zhang, "NHPP software reliability and cost models with testing coverage, " in European Journal of Operational Research, vol. 145, no. 2, pages. 443–454, Mar.2003.

[24] A. Podgurski, D. Leon, P. Francis, M. Minch, J. Sun, B. Wang and W. Masri, "Automated support for classifying software failure reports," in Proceedings of 25th ICSE, Portland, OR, May 2003.

[25] A. Podgurski, W. Masri, W., Y. McCleese, F. G. Wolff, and C. Yang, "Estimation of software reliability by stratified sampling," in ACM Transactions on Software Engineering and Methodology 8, 9 (July, 1999), pages 263-283.

[26] ROME Project. http://java.net/projects/rome/

[27] D. S. Rosenblum, E. J. Weyuker, "Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies, " in IEEE Trasaction on Software ngineering, Vol. 23, No. 3, Mar. 1997, pages 146-156.

[28] B. Settles. "Active learning literature survey,"Technical Report 1648, University of Wisconsin- Madison. 2009.

[29] Spring. Spring Framework 3.0.2.RELEASE, www.springframework.org.

[30] J. Steven, P. Chandra, B. Fleck, A. Podgurski, "jRapture: A Capture/Replay tool for observation-based testing," SIGSOFT Softw. Eng. Notes, vol. 25, 2000, pages. 158-167.

[31] J. Tian and J. Palma, "Test workload measurement and reliability analysis for large commercial software systems," Annals of Software Engineering 4 (1997), pages 201-222.

[32] M. Z. Tsoukalas, J. W. Duran, S. C. Ntafos. "On some reliability estimation problems in random and partition testing," Software Engineering, IEEE Transactions on 1993, 19(7), pp. 687-97.

[33] E. J. Weyuker, "On Testing Non-Testable Programs, " in The Computer Journal (1982) 25 (4), pp. 465-470

[34] E. J. Weyuker, "Using failure cost information for testing and reliability assessment," ACM TOSEM 1996; 5(2): 87-98.

[35] Xerces2 Project. http://xerces.apache.org/xerces2-j/